An operational formal definition of PROLOG

Pierre Deransart, Gérard Ferrand

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 598

AN OPERATIONAL FORMAL DEFINITION OF PROLOG

Pierre DERANSART
Gérard FERRAND

Décembre 1986

Une Sémantique Formelle Opérationnelle de PROLOG

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cédex
Tél. : (1) 39 63 55 36

Gérard FERRAND
Université d'Orléans
Faculté des Sciences
B.P. 6759
45067 ORLEANS Cédex 2
Tél. : 38 63 22 16

Résumé :

Nous présentons dans ce rapport une nouvelle sémantique formelle opérationnelle adaptée à la description des dialectes PROLOG "déterministes" (de type PROLOG Marseille ou Edinbourgh). En particulier, cette sémantique peut être utilisée pour décrire complètement le futur PROLOG standard.

Cette sémantique consiste en un interprèteur abstrait écrit en PROLOG "pur" (avec négation) dont la sémantique est purement déclarative. On peut donc comprendre l'interprèteur sans se référer à aucun dialecte particulier. Bien que fondée sur PROLOG, cette sémantique ne se mord pas la queue. La spécification obtenue n'est pas directement executable.

Son avantage principal réside dans le fait qu'elle est compréhensible par toute personne qui connait au moins un dialecte et qu'elle satisfait les critères d'une bonne spécification formelle (abstraction, simplicité, vérifiable, utilisable pour construire des outils de validation, en particulier obtenir une spécification exécutable).

Mots clés :

Programmation en Logique, Spécification Formelle, Sémantique Opérationnelle, Sémantique de PROLOG, Standardisation de PROLOG.

An Operational Formal Definition of PROLOG

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cédex
Tél. : (1) 39 63 55 36

Gérard FERRAND
Université d'Orléans
Faculté des Sciences
B.P. 6759
45067 ORLEANS Cédex 2
Tél. : 38 63 22 16

Abstract :

We present in that report a *new formal (operational) semantics* well adapted to describe PROLOG's deterministic dialects (Marseille-Edinbourgh-like dialects [Col 82, PWB 84]). In particular it could be used in the standardization work of PROLOG [DF 86b].

This semantics is based on an abstract interpreter written in a "pure" PROLOG style (with negation) whose semantics is itself essentially declarative. Thus it can be understood without any reference to any particular dialect, since the negation is treated declaratively (This point will not be developped completely inside this paper). Such a semantics although written in PROLOG, does not depend on itself. It is not an executable specification.

One of its advantages is to be readable without backgrounds by users of a PROLOG dialect It satisfies also the criteria of a good formal specification of a programming language:

- To be of a very high level, but as simple and understandable as possible.
- To be able to describe a large variety of dialects.
- To describe with the same formalism the whole language.
- To be able to be certified, using validation methods.
- To make easy the production of validation tools ,in particular of a runnable specification.

Keywords :

Logic programming, Formal Specifications, Operational Semantics, PROLOG Semantics, PROLOG Standardization.

An Operational Formal Definition of PROLOG

Pierre DERANSART
INRIA
Domaine de Voluceau
B.P. 105 - Rocquencourt
78153 LE CHESNAY Cédex
Tél. : (1) 39 63 55 36

Gérard FERRAND
Université d'Orléans
Faculté des Sciences
B.P. 6759
45067 ORLEANS Cédex 2
Tél. : 38 63 22 16

12 Dec. 86

1) Introduction:

We present in that report a *new formal (operational) semantics* well adapted to describe PROLOG's deterministic dialects (Marseille-Edimburgh-like dialects [Col 82, PWB 84]). In particular it is used in the standardization work of PROLOG [DF 86b].

This semantics is based on an abstract interpreter written in a "pure" PROLOG style (with negation) whose semantics is itself essentially declarative. Thus it can be understood without any reference to any particular dialect, since the negation is treated declaratively (this point will not be developped completely inside this report). Such a semantics although written in PROLOG, does not depend on itself. Although it is known from the folklore that logic programming can be used as a specification language, it is rarely realized in most of the attempts to use logic programming to specify languages, systems or known PROLOG dialect. Most of the realizations use some unspecified feature (built-in predicate, cut or negation by failure, implicit interpretation strategy), because they are more devoted to simulate an interpreter rather to specify it. Thus this "PROLOG" semantics is new because of the following points :

- It is completely declarative and formal.
- It has a simple syntax contained in every prolog dialects. However it does not depend on itself. To overcome this difficulty, one uses the notions of relative denotation and declarative semantics of the negation. For these reasons it is not an executable specification.
- It is of higher level than all the others expressed in logic programming style: It uses the search-tree as semantical data.

One of its advantages is to be readable without major difficulties by users of a PROLOG dialect. It satisfies also the criteria of a good formal specification of a programming language:

- To be of a very high level, but as simple and understandable as possible.
- To be able to describe a large variety of dialects, with the same level of abstraction.
- To describe with the same formalism the whole language.
- To be able to be certified, using validation methods.
- To make easy the production of validation tools of the language implementations.

These points will be discussed in the last section.

This semantics is also interesting because it illustrates the power of logic programming to make specifications. It seems to us that logic programming is generally considered as "impure" executable specification. Our purpose is to show that logic programming may also be used as a perhaps low level but full specification language. Work is actually in progress to apply this idea to the whole project of PROLOG standard.

The report is organized as follows: section 2 defines the semantics of PROLOG dialects using mostly known concepts, but organized in an original way. In sections 3 and 4 we define the formal specification dialect: its semantics (3) and the basic objects used in it (4). Section 5 gives a sample of the formal definition for a piece of PROLOG with cut and geler (freeze). Discussion will be held in the section 6.

2) Semantics of the PROLOG dialects :

Most of the basic concepts are known. They are:
- denotation of a logic program [Fer 85].
- proof tree [Cla 79, DM 85].
- unification [Rob 65].
- search tree [Cla 79, AvE 82, Llo 84].

But their presentation has been modified in order to make them independent of their historical origin, as proofs by refutation, and to fit exactly with the concepts that users of an interactive prolog interpreter may observe (proof tree, answer substitution). Moreover, as such they permit to simplify the presentation of the logic programming formalism used in the formal definition.

a) Abstracts syntax :

This syntax is sufficient to describe the (abstract) syntax of most of the usual PROLOG dialects. It is as follows:
- Terms (without types). Variables symbols are concretely distinguished from other symbols by begining with a capital letter.
- Atoms (atomic formulae) built with predicate symbols and terms. A litteral is a positive or negative atom (denoted not a).
- Clauses $a_0 \Leftarrow a_1, ..., a_n$ where a_0 is the head, $a_1, ..., a_n$ the body : a list of litterals (the order of the litterals in the body is significative). true is the empty body.
- Program : list of clauses (significative order) regouped into packets.
- The considered goals are restricted to one litteral without loss of generality.

b) Logical semantics of a logic program P, Its denotation :

The logical semantics of a logic program P (without negation) is the set of all its atomic logical consequences. We denote DEN(P) this set, also called the denotation of P. This approach is different from all the others [Cla 79, AvE 82, Llo 84] which use the Herbrand model. It has been justified in [Fer 85, DF 86a], but the properties of this semantics have been studied in [Cla 79, Fer 85]. They are:
- DEN(P) *is also a model of* P *(also the least in the termal interpretations).*
- DEN(P) *is closed under substitutions.*
- *Given a goal* b , *a PROLOG interpreter build answer substitutions* σ *such that* $\sigma b \in$ DEN(P).

Examples :

$P_1 =$ { plus (o, X, X) $\Leftarrow$.
plus (sX, Y, sZ) $\Leftarrow$ plus (X, Y, Z). }

$DEN(P_1) = \{ \text{plus}(s^n o, t, s^n t) \mid \forall\ n \geq o, \forall\ t \text{ term} \}$

Answer substitutions :

plus (so, X, Y) ? $\sigma = \{Y \leftarrow sX\}$
plus (X, Y, so) ? $\sigma_1 = \{X \leftarrow o, Y \leftarrow so\}$
$\sigma_2 = \{X \leftarrow so, Y \leftarrow o\}$

c) Constructive semantics (proof-tree):

The constructive semantics formalizes the notion of proof using the so called proof trees (also in [Cla 79]). A proof tree is a finite tree whose nodes are labeled by atoms (or litterals in the case of the negation), the root is an instance of a goal, all leaves are labeled by true and all the elementary subtrees are instances of a clause.

Such a tree is the representation of a proof of its root (atomic theorem), using the clausal implications.

Example : (Program P_1)

- plus (so, X, sX)
- plus (o,X, X)
- true

It is easy to show the following completness result:

DEN(P) *is exactly the set of the proof -tree roots* [Fer 85, DF 86a].

The constructive semantics of a program P is the set of all the proof-tree roots, i.e. its denotation. It is declarative.

d) Non deterministic operational semantics (search tree): (procedural semantics in[Cla79])

The answer substitutions are constructed by deriving from a given goal a proof-tree in a top down manner. The construction uses some unification algorithm [Rob 65] and takes advantages of the properties of the unifiers (unique most general unifier- MGU).

We recall here the non deterministic algorithm deriving the proof-trees.

b : goal
P : program
T : proof-tree
T : = b ;
while it exists a leaf not labeled by true in the tree T
do *choose* l *some leaf different from true ;*

choose c: ao ⇐ a1,..., an *some clause with variables different from those appearing in T;*
σ : = M G U (l, a_o) ;
T : = σT [σl ← σa_o]
σa_1 ... σa_n

The constructive semantics is complete. It means that for each σ such that $\sigma b \in$ DEN(P), σb is an instance of the root of some proof-tree obtained by this construction.

Search-tree (also as SLD-tree in [AvE 82, Llo 84])

The non determinism of this algorithm comes from two points :

- non determinism of type l : choice of a leaf ("computation rule" in [Llo 84]). We refer to the l-strategy or l-choice function.
- non determinism of type c : choice of a clause. We refer to the c-strategy or c-choice function.

The "standard l-strategy" consists in fixing the first choice by building the proof-tree choosing the first left-most leaf different from true.

Any other strategy can be defined using the notion of search-tree.

Given a program P, a goal b, and a deterministic strategy of type I (choice of the leaves ir the proof trees), we can associate to P a unique search-tree by the following way :

- each node is labeled by a list of literals (empty list = true), one of them being distinguished (choosen) by the type I choice function.

- from each node as many arrows are issued as there are heads of clauses in P which can be unified with its choosen atom.

- if $I = b_1, ..., \underline{b_i}, ..., b_n$ is the node, b_i beeing the distinguished literal, and there is q clauses in P whose head is unifiable with b_i, presented in the order [1..q], noted $a_j \Leftarrow c$, $1 \leq j \leq q$, then the node I has q sons : $n_1, ..., n_j, ..., n_q$ such that :

  n_j =success if I = b, b and a_j unifiable, c = true

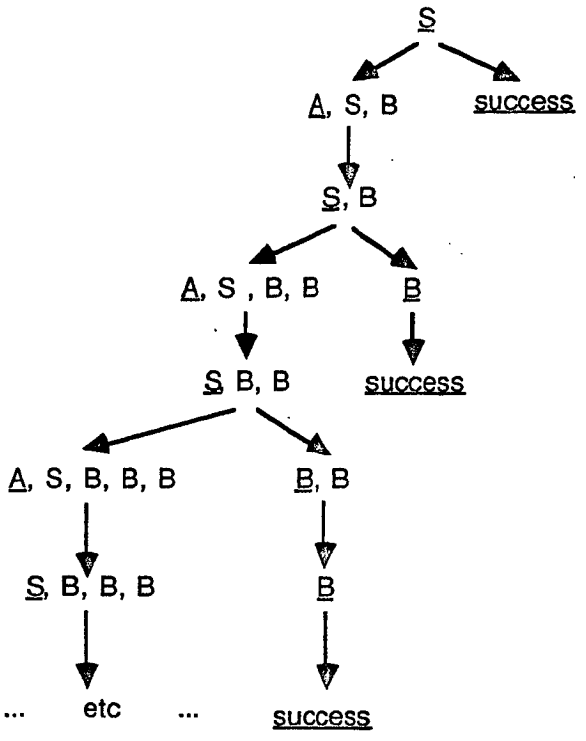
  $n_j = \sigma(b_1, ..., c, ..., b_n)$, σ = MGU (b_i, a_j) otherwise.

Example :

Given the program :

1. S $\Leftarrow$ A, S, B.
2. S $\Leftarrow$.
3. A $\Leftarrow$.
4. B $\Leftarrow$.

with the standard I-strategy and given the goal S, the search-tree is the following (the distinguished atom are underlined) :



The search-tree describes all the ways to build all the proof-trees, after giving a l-strategy . A search-tree can be infinite. To any partial finite path issued from the root in the search-tree it corresponds a partial proof-tree. The proof-tree is complete if the leaf of the path is labeled by success.

e) Deterministic operational semantics:

Given a search-tree, to fix the non determinism of type c consists in giving an order of tree-walk. The tree-walk defines the order in which the answer substitutions are obtained.

Thus, the operational semantics of a logic program P, given a goal b, is the search-tree derived from the goal with its walk order. This semantics is sufficient to take into account definite Horn clauses programs.

Also this semantics is particularly well suited to explain the effects of control features like "cut" (dynamic modification of the tree) and delaying primitives (non standard l-choice) [DF 86a,b]. Because of the search-tree modifications, the operational semantics of most of the dialects cannot be described by a single search-tree, but by a (possibly infinite) set of partia search-trees corresponding to some significant steps of walk. Thus the semantics of a logic program is the set of partial search-trees obtained following the constructive walk order. Now it should be clear that various semantics could be defined depending on the selected set of partial search-trees corresponding to the semantics.

Remark that this semantics is especially interesting in the following limit cases: if the search-trees are always finite, then the semantics may contain the unique finite search-tree and it is sufficient to describe the whole semantics of the program (with its goal). On the other hand, if the search-tree is infinite without finite success branches then all partial finite search-trees must be in the semantics.

In the sequel a specific semantics will be choosen.

3) **Semantics of the formal specification** :

The formal specification language is a very simple logic programming dialect which we describe now together with its semantics.

a) Abstracts syntax :

As in section 2-a, but the order of the clauses in the packets as the order of the litterals in the body of the clauses is irrelevant.

b) Relative denotation :

The notion of denotation is sufficient to describe the semantics, but to deal with the PROLOG dialects which contain built-in predicates, an extended notion of denotation is needed: the relative denotation.

Given a set E of atoms, we denote **E** the set of clauses $\{A \Leftarrow . | A \in E\}$. The denotation of the logic program P relative to E , denoted DEN(P|E) is the denotation $DEN(P \cup \mathbf{E})$ of the logic program $P \cup \mathbf{E}$.

For example consider a predicate $\mathbb{N}$-**plus**(X, Y, Z) which satisfies Z = X+Y (on natural intergers $\mathbb{N}$). **E** contains all the clauses like :

$\mathbb{N}$-**plus**(X, Y, Z) $\Leftarrow$. for all X, Y, Z integers such that Z = X+Y.

The logical semantics of a logic program P using this predicate (not defined in P) is exactly $DEN(P|E) = DEN(P \cup \mathbf{E})$.

c) Contructive semantics : proof-trees with negation

In the formal definition we want to avoid the use of control features -like the cut- which can be only understood by reference to some interpretation strategy. But in order to preserve a good expressiveness of the formal specification language we will make use of the negation which may be nicely undestood declaratively. Various logical semantics can be given to logic programs with negation [Llo 84, Del 86]. No one is satisfactory on the programmer point of view. So we have choosen to use a negation with a clear declarative semantics, closed to the usage of negation by failure implemented in most of the PROLOG dialects. A similar notion has been introduced in [ABW 86] and has been treated completely in [DF 86d]. It is based on a generalization of the notion of proof-tree with negative literals, whose principles are recallec here briefly.

The generalization is the following :

A leaf of a proof-tree can also be a negative literal "not a" iff no instances of "a" is the root of a proof-tree.

The intuition behind this "definition" is to preserve the declarative notion of denotation and to fit with the commonly accepted idea of the negation in the PROLOG folklore : an atom is false if no instance can be proven true. Thus a negated atom will be true iff it cannot be root of any proof-tree at all, i.e. no instances are in the denotation.

Such definition can be understood only if the programs are stratified (we use the same name as in [ABW 86] but with a slightly different formal definition). Intuitively -only a simplified notion is recalled here- a program (thus the specification) is stratified by levels levels 0 contains packets of clauses without negative literals. At level 1 only packets using negative literals of level 0 are included, etc.. level n contains packets with negative literals of lower levels. With such a stratification, it is easy to define constructively, step by step, the set of all proof-trees with negation of a given logic program with negation.

The constructive semantics of a program is the set of all the proof-tree roots with negation. We call it again the denotation of P : DEN(P).

d) Declarative semantics of logic programs with negation

The declarative semantics of a logic program P with negation is defined as the union of DEN(P) and NEG(P) where DEN is as in b) and NEG(P) is the set of all literals "not a" such that no instance of a is in DEN(P).

This semantics does not have a very simple logical semantics. A complete treatment of this aspect is presented in [DF 86d] . But it has a simple intuitive logical semantics which will be used in the specification. Given a logic program P with negation, we denote P^* the program P in which all the litterals "not a(X)" in the body of the clauses have been replaced by "$\forall$X not a(X)" (or "not $\exists$X a(X)"). Thus DEN(P) is a model of P^*. This shows that this semantics is consistent (It is not the case of the closed word assumption). Moreover, this kind of program "transformation" corresponds to the usual intended meaning of the negative litterals in logic programs.

This last remark shows the restrictions over the negation we will make in the formal definition language in order to preserve its clarity. A negative literal may appear with variables in the body of a clause, but not all variables play the same role. In fact negative litterals appear in the context of proof-trees in which not all possible instances of the program clauses are effectively used. It will be clear in the formal definition that some variables will be always replaced by ground terms. For example in the specification the semantical objects (programs, search-trees, L-variables i.e. variable in the described languages) are representec by ground terms. Thus in the program "buildlist" (section 4) the negative literal "not L-unifiable (H, A)" in the second clause has an intended meaning "not L-unifiable of some head H and some atom A when H and A are atoms of L ", i.e. of the specification point of view H and A are always ground, thus the meaning of the stratified program describing the specification becomes clear. Note that it is not the case in the program c-choice (section 4 - second clause) in which "not son (T, _)" is used with an anonymous variable. In this case the intended meaning is that T is ground and not "$\exists$X son (T, X)", which fits with the intuitive logical semantics.

In [DF 86d] the links between this kind of negation and the usual implementation in standard interpreters of the negation as failure are completely developped. This shows that the restricted use of the negation inside the specification will permit to transform it into a runnable specification without major difficulties, using the known concept of "safe negation" [Llo 84, DF 86d].

4)**Basic elements of the formal definition**:

The formal definition will be given by a logic program (definite Horn clauses program with negation) describing the search-tree associated to some given program P together with a goal. Thus the operational formal semantics of a program P and a goal g is the declarative semantics of this program.

More precisely it corresponds to the relative denotation of the relation :

Semantics (P, G, T)

where **P** is a logic program, **G** a goal, **T** a partial search-tree issued from **G** which is complete (if **T** is finite) or limited to some last constructed finite success branch following the tree-walk order of **T** (thus only programs and goals such that it exists at least one answer

substitution obtained following the program strategy or with finitely failed search-tree will be considered in this sample of formal definition).

The denotation of **Semantics** contains all partial search-trees corresponding to a success branch obtained by the specific strategy of P and the complete one if it exists a finite search-tree. For simplicity we have restricted the semantics considering only to steps corresponding to success pathes or complete finite search-trees. This kind of limitation coulc be easely suppressed to take into account infinite computations without any success path.

It is worth noticing that the intended semantics is very simple : the (non)standard strategies of P can be described using only a logic program without references to any strategy.

To understand the formal semantics one must know the used data structures : relative denotation, programs and search-tree.

L-language

It is supposed that predicates and functions of the described language are written in a defined language denoted L. It contains all possible symbols, including variables names which are considered as constants in the formal semantics. In the following it will be necessary to rename variables.

We will use, as in [JM 84] integers to rename denumerable sets of variables. The used integers will be represented in lists denoting in a Dewey like notation the nodes of the search-tree also.

Relative denotation

As it has been shown, the relative denotation is a way to give a meaning to the following predicates which will be used in the formal definition without logic description.

L-var(X) iff X is a variable in L.

L-term(X) iff X is a free term in L.

L-ground(X) iff X is a ground term in L.

L-instance(T_1, S, T_2) iff T_2 is an instance of T_1 by the substitution S.

L-unify (T_1, T_2, S) iff S exists and S is the MGU of T_1 and T_2.

L-rename(I, T_1, T_2) iff T_2 is T_1 with renamed variables using the integer (or indice) I.

Notice that we could define :

L-unifiable(T_1, T_2) $\Leftarrow$ **L-unify**(T_1, T_2, S).

We denote by = the L-syntactic equality ($T_1 = T_2$ iff T_1 and T_2 are the same L-terms).

Structure of a program P

P is a list of clauses. The clauses defining a predicate are grouped in a list called **packet**. This

later predicat will not be described here. Its meaning is the following :

packet(P, I, A, Q) iff **Q** is an instance of the packet of **P** corresponding to the predicate of the atom **A**, renamed using **I**. **Q** is a list of clauses whose variables are all distinct of those of **P**. If **A** has no corresponding packet, thus **Q** = nil (empty list).

structure of the search trees :

A tree is :

a	node (I, G)	I	list of integers built with o and succ (successor).
		G	goal = list of literals.
			lists are doted. Empty list is nil.
or	mk-tree (N, F)	N	node
		F	list of trees.

Note that all the nodes in a search-tree are different.

A goal is represented by a list of literals denoted differently:

true (empty goal)

A + L literal A, L goal.

Search-trees are constructed by extensions or modified (the cut modifies and extend a search-tree).

In the whole program a tree argument is denoted by a term T-N where T is a tree (or a list c trees) and N a node in T. This notation avoid the use of two arguments where it is unneccessary. Anonymous variables, when they are not necessary to understand the definition, are written _ as in [PWB 84].

The program uses tree-walk functions defined as follows :

if N1 and N2 are nodes in T, then:

first-son($T - N_1, N_2$) iff N_2 is the first son of N_1.

son($T - N_1, N_2$) iff N_2 is one of the sons of N_1.

brother($T - N_1, N_2$) iff N_2 is the first brother of N_1.

father($T - N_1, N_2$) iff N_2 is the father of N_1.

if N is a node then:

N **in** T iff N is a node of T.

if G is a goal and A an atom then:

tail(G_1, G_2) iff G_2 is the list rest of G_1.

head(G, A) iff A is the head atom of G.

Logical definition :

first-son(mk-tree(N_1 , N_2.L)-N_1 , N_2) ⇐ **not** N_2 = mk-tree(_, _) .

first-son(mk-tree(N_1,mk-tree(N_2, F).L) - N_1, N_2) ⇐ .

first-son(mk-tree(N,F)-N_1, N_2) ⇐ **first-son**(F-N_1, N_2) , **not** N = N_1 .

first-son(T.L -N_1, N_2) ⇐ **first-son**(T-N_1, N_2) .

first-son(T.L -N_1, N_2) ⇐ **first-son**(L - N_1, N_2) .

son(T -N_1 , N_2) ⇐ **first-son**(T -N_1 , N_2) .

son(T -N_1 , N_2) ⇐ **first-son**(T -N_1 , N_3) , **brother**(T -N_3 , N_2) .

brother(N_1.N_2.L -N_1, N_2) ⇐ **not** N_2 = mk-tree(_, _) .

brother(mk-tree(N_1, F).N_2.L -N_1, N_2) ⇐ **not** N_2 = mk-tree(_, _) .

brother(N_1.mk-tree(N_2, F).L-N_1, N_2) ⇐ .

brother(mk-tree(N_1, F_1). mk-tree(N_2, F_2).L-N_1, N_2) ⇐ .

brother(mk-tree(N, F)-N_1, N_2) ⇐ **brother**(F-N_1, N_2) .

brother(T.L-N_1, N_2) ⇐ **brother**(T-N_1, N_2) .

brother(T.L-N_1, N_2) ⇐ **brother**(L-N_1, N_2) .

father(T-N_1, N_2) ⇐ **son**(T-N_2, N_1) .

N **in** N ⇐ .

N **in** T.L ⇐ N **in** T .

N **in** T.L ⇐ N **in** L.

N **in** mk-tree(N, F) ⇐ .

N **in** mk-tree(N_1, F) ⇐ N **in** F .

tail(A+G, G) ⇐ .
head(A+G, A) ⇐ .

5) **A sample of formal specification**:

```
Semantics(P, G, T) ⇐
        buildtree(P,   mktree(node(o.nil, root), node(o.nil,G).nil)-node(o.nil,G) ,   T) .
```

where if **P** is a logic program and T_1 is a partial search-tree whose current node is N_1, then **buildtree(P**, T_1-N_1, T_2 -N_2) is true iff T_2 is the partial search-tree obtained by a constructive tree-walk of T_1 from N_1 until a success node has been reached (N_2 = node (I, true)), or a complete finite search-tree has been obtained (N_2 = node (o.nil, root)).

```
buildtree(P,  T-N,  T-N) ⇐   N = node(I, true) .

buildtree(P,  T-N,  T-N)  ⇐  N = node(o.nil, root).

buildtree(P,  T1-N1, T2)  ⇐  c-choice( T1-N1,  N2) ,
                             treatment(P,  T1-N2,  T3) ,
                             buildtree(P, T3,  T2) .
```

The tree construction is "interrupted" as soon as a success node is reached or the constructior is completed. In the first case, it will continue (not exclusive cases). The axioms can be read axiomatically or operationally : choose a node (N_2) which is always a leaf, make a treatment (the current node remains unchanged), continue the construction.

Only a very restricted part of a possible specification is given hereby. The treatments are thus very simple ones. Morever, we will suppose that the intended c-strategy of a program corresponds to a top-down left to right search-tree-walk.

```
c-choice(T, N)          ⇐  first-son(T, N) .

c-choice(T, N)          ⇐  not son(T, _) , brother(T, N) .

c-choice(T, N)          ⇐  not son(T, _) , not brother(T, _) ,
                           re-c-choice(T, N) .

re-c-choice(T-N1, N2)   ⇐  not brother(T-N1, _) , father(T-N1, N3) ,
                           re-c-choice(T-N3, N2) .

re-c-choice(T, N)       ⇐  brother(T, N) .

re-c-choice(T-N, N)     ⇐  not brother(T-N, _) , not father(T-N , _) .
```

One may show without difficulty that in the case of this sample definition this strategy is sufficient, because of the specific form of the trees .

```
treatment(P, T-N,   T-N)     ⇐  N = node(_, true) .

treatment(P, T-N, T -N)      ⇐  N = node(o.nil, root) .

treatment(P, T-N, T-N)       ⇐  N = node(I, G), not G = root ,
                                not I-choice(G, A) .

treatment(P, T1 -N, T2)      ⇐  N = node(I, G), I-choice(G, !(J)) ,
                                treat-cut(T1-N, T2, J) .

treatment(P, T1-N, T2)       ⇐  N = node(I, G) , I-choice(G, A) , not A = !(J) ,
                                expand(P, T1-N, A, T2) .
```

if N_1 is a leaf of T_1 or N_1 is the root then **treatment**(P, T_1-N_1, T_2-N_2) is true iff T_2 is the new search-tree (after eventual modification) with N_2 as current node .

The treatment consists in firstly choosing a distinguished literal of the node label and secondly either expanding it or modifying it to take in account the effect of the cut. The intended strategy is the standard one. The effect of the "cut" is to suppress all not visited nodes following (in the tree-walk order) all nodes in a path from the current node to the father of the node where the current cut (!(J)) has been labeled by J. As the modifications do not affect already visited search-tree parts, we still have $N_2 = N_1$.

```
I-choice(G, A)  ⇐  head(G, A) .
```

This is the standard choice function of type I (first literal in the goal list). To take in account a "geler" (freeze) predicate could give the following description :

```
I-choice(A+L, B)              ⇐  not A = geler(_, _),  first-unfrozen-geler(L, B) .

I-choice(A+L,  A)             ⇐  not A = geler(_, _),  all-geler-frozen(L) .

I-choice(geler(X, B)+L,  A)   ⇐  L-var(X) ,   I-choice(L, A) .

I-choice(geler(X, A)+L,  A)   ⇐  not L-var(X) .

first-unfrozen-geler(L,  A)   ⇐  conc(L1,  geler(X, A)+L2,  L) ,
                                 all-geler-frozen(L1) ,
                                 not L-var(X) .
all-geler-frozen(true)        ⇐  .

all-geler-frozen(A+L)         ⇐  not A = geler(_, _) ,   all-geler-frozen(L) .

all-geler-frozen(geler(X,  A)+L)  ⇐  L-var(X) ,  all-geler-frozen(L) .
```

All frozen goals are unfrozen following the order of freezing. **conc** is the list

concatenation on goal lists.

If **P** is a logic program, $\mathbf{N_1}$ the current node of $\mathbf{T_1}$, **A** the distinguished literal in $\mathbf{N_1}$ then **expand**(**P**, $\mathbf{T_1}$-$\mathbf{N_1}$, **A**, $\mathbf{T_2}$-$\mathbf{N_2}$) is true iff $\mathbf{T_2}$ is the new tree after expansion. $\mathbf{N_2}$ (=$\mathbf{N_1}$) corresponds to a new sub-tree : mk-tree ($\mathbf{N_1}$, F) where F are the new sons, but which is reduced to $\mathbf{N_1}$ if $\mathbf{N_1}$ is a failure node. When a new node is created, the cuts it contains have to be labeled by the name of the node itself.

```
expand(P, N-N, A, N-N)                    ⇐  N = node(I, _) , packet(P, I, A, nil) .

expand(P, N-N, A, mk-tree(N, F)-N)        ⇐  N = node(I, _) ,
                                             packet(P, I, A, Q) , not Q = nil ,
                                             buildlist(Q, N, A, F,o) .

expand(P, mk-tree (N, F1)-N1, A, mk-tree(N, F2)-N2)  ⇐
                                             expand(P, F1-N1, A, F2-N2) .

expand (P, A1.L-N1, A, A2 .L-N2)          ⇐
                                             expand(P, A1-N1, A, A2-N2) .

expand(P, A.L1-N1, B, A.L2-N2)            ⇐  expand(P, L1-N1, B, L2-N2) .

buildlist(nil, N, A, nil, I)              ⇐  .

buildlist(clause(H, C).Q, N, A, F, I)     ⇐  not L-unifiable(H, A) ,
                                             buildlist(Q, N, A, F, I) .

buildlist(clause(H1, C1). Q, N, A, node(I.J, G2).F, I)  ⇐  N = node(J, G) ,
                                             L-rename(I.J,clause(H1,C1), clause(H2,C2)),
                                             L-unify(H2, A, S) ,
                                             flagcut(C2, I.J, C3),
                                             replace(G, A, C3, G1) ,
                                             L-instance(G1, S, G2) ,
                                             buildlist(Q, N, A, F, succ (I)) .

replace(L1, A, L2, L3)                    ⇐  conc(X, A.Y, L1), conc(L2, Y, Z) ,
                                             conc(X, Z, L3) .

flagcut(true, I, true)                    ⇐  .
flagcut(! + L1, I, !(I)+L2)               ⇐  flagcut(L1, I, L2) .
flagcut(A+L1, I, A+L2)                    ⇐  not A = !, flagcut(L1, I, L2).
```

if N_1 is a leaf node of T_1 whose chosen atom is I(J), then **treat-cut**(T_1-N_1, T_2-N_2, J) is true iff T_2 is the modified tree T_1. It has a new son node issued from N_2 (= N_1).

treat-cut(N-N, mk-tree(N,node(o.I,G_2).nil)-N, I) ⇐ N = node(I,G_1) ,
tail(G_1,G_2) .
(depends on the I-trategy: here it is the standard one)

treat-cut(mk-tree(N, F_1)-N_1, mk-tree(N, F_2)-N_2, I) ⇐
treat-cut(F_1-N_1, F_2-N_2, I) .

treat-cut(T_1.L-N_1, T_2.nil-N_2, I) ⇐ N_1 **in** T_1 , **cut-inside**(I, T_1) ,
treat-cut(T_1-N_1, T_2-N_2, I) .

treat-cut(T_1.L-N_1, T_2.L-N_2, I) ⇐ N_1 **in** T_1 , **not cut-inside**(I, T_1) ,
treat-cut(T_1-N_1, T_2-N_2, I) .

treat-cut(T.L_1-N_1, T.L_2-N_2, I) ⇐ N_1 **in** L_1 , **treat-cut**(L_1-N_1, L_2-N_2, I) .

cut-inside(I, node(J, G)) ⇐ **cut-member**(I, G) .

cut-inside(I, mk-tree(node(J, G), F)) ⇐ **cut-member**(I, G) .

cut-member(I, I(I)+L) ⇐ .

cut-member(I, A+L) ⇐ **cut-member**(I, L) .

6)**Discussion and conclusion**:

We have presented a formal semantics for PROLOG dialects which seems to satisfy the criteria of a good formal specification method as given in the introduction:

-It is of high level because without any reference to any abstract machine. Also it does not use any too low level programming language. The logical notions it uses are simple and known by most of the PROLOG programmers.

-It can be used to describe with the same formalism most of the existing PROLOG dialects. Without major difficulties it is possible to take into account the following features:

. cut
. geler(or freeze), wait (any non standard deterministic strategy)
. assert, retract
. constraints, dif, delayed negation
. infinite terms : Rational infinite terms can be added to the language L whose treatment can be descibed in a logical specification using finite terms only. Thus even if the described dialect containing rational infinite terms does not have any logical semantics (in the sense of section 2), the specification still have one.
. assignment, global variables
. escape or block mechanisms
. arithmetics and other built-in predicates. These are in fact not described by clauses in a program, however they belong to the described language L and we want to include their description in the same formalism. It is partially possible. As an example we give here a description of a reversible built-in predicate **plus** using a semantical predicate **L-eval** with two arguments: some goal (chosen atom) and the "computed" answer substitution S.

L-eval(plus(X, Y, Z), S) ⇐ **ℕ-number**(X), **ℕ-number**(Y), **L-var**(Z),
ℕ-plus(X,Y,Z1), **L-unify**(Z, Z1, S).

L-eval(plus(X, Y, Z), S) ⇐ **ℕ-number**(X), **ℕ-number**(Z), **L-var**(Y),
ℕ-plus(X,Y1,Z), **L-unify**(Y, Y1, S) .

L-eval(plus(X, Y, Z), S) ⇐ **ℕ-number(Y)**, **ℕ-number(Z)**, **L-var**(X),
ℕ-plus(X1,Y,Z), **L-unify**(X, X1, S) .

L-eval(plus(X, Y, Z), ∅) ⇐ **ℕ-number**(X), **ℕ-number**(Y), **ℕ-number**(Z),
ℕ-plus(X,Y,Z).

This description uses a not described predicate **ℕ-number**. Such predicate will be supposed to have a clear denotation which can be easely defined under the asumption that the domain ℕ of the natural integer is wellknown enough. Thus the denotation of **ℕ-number** contains all atoms

ℕ-number(i) where i is an interger of ℕ.

ℕ-plus is as in section 3.

-As the formal definition is a logic program the validation methods (correctness, completeness, termination, use of the negation, ... [DF 86c]) could be applied to validate the formal definition itself.

-As the formal definition is a logic program included in every dialect, it should be rather

simple to derive from the specification various validation tools like a runnable specification.

This kind of formal definition can be compared with others logical semantics of logic programming such as in [Mos 81] and [MR 86]. Moss's approach describes proof-trees construction in place of search-trees construction. It consists in giving a validating program (i.e. a PROLOG program simulating an interpreter) and thus it is less declarative. Martelli and Rossi approach can be considered as a logic runnable specification of a denotational one and it uses a kind of abstract machine. Considering other kind of semantics like in [JM 84, Fra 85, Nor 86] the denotational semantics, we could say that our semantics is less concise but easier to read, especially for people knowing some PROLOG dialects.

We have shown how a logic program of about sixty clauses could model the formal semantics of logic programs with cut and a non standard strategy. The size of a complete description of a dialect or a standard would increase in proportion with the number of considered primitives, but preserving the modularity and the clarity of the formal definition. I can be improved also by choosing simpler data structures as ramifications in [Fer 85]. Some questions, as input/outputs or errors handling, have not been investigated here. Some adaptation of the presented semantic model would be necessary, but are still possible by using the same level of abstraction.

AKNOWLEDMENTS

We are indepted to Gilles Richard for his comments on the earlier drafts which helpec to improve this presentation.

BIBLIOGRAPHY

[AvE 82] K. R. Apt, M. H. Van Emden : Contributions to the theory of logic programming. JACM V29, N° 3, Jul. 1982 pp 841-862.

[ABW 86] K.R. Apt, H. Blair, A. Walker: Toward a Theory of Declarative Knowledge. LITP res. report 86-10, Fev 1986.

[Cla 79] K. L. Clark : Predicate Logic as a Computational Formalism. Res. Mon. 79/59 TOC. Imperial College. Dec. 79.

[Col 82] A. Colmerauer: Prolog II: Manuel de reference et modele theorique, GIA, Univ. of Marseille, 1982

[Del 86] J.P. Delahaye : Sémantique logique et dénotationnelle des interpréteurs PROLOG. Note IT n° 84. University of Lille. 1986.

[DF 86a] P. Deransart, G. Ferrand: Initiation a PROLOG, Concepts de base. Pub. du Lab

d'Informatique, Univ. of Orleans, June 1986.

[DF 86b] P. Deransart, G. Ferrand: An Operational Formal Definition of PROLOG. A note for the AFNOR-BSI group on PROLOG normalization. 10/05/86, BSI PS/112.

[DF 86c] P. Deransart, G. Ferrand : Logic Programming, Methodology and Teaching. Actes du Seminaire 1986, CNET Tregastel mai 1986, pp75-90 (english version available at INRIA-Rocquencourt).

[DF 86d] P. Deransart, G. Ferrand : Programmation en Logique avec négation : présentation formelle. Publication du Laboratoire d'Informatique, University of Orléans (to appear).

[DM 85] P. Deransart, J. Maluszynski : Relating logic Programs and Attribute Grammars. Journal of Logic Programming 1985, 2, pp 119-155.

[Fer 85] G. Ferrand : Error Diagnosis in Logic Programming, an Adaptation of K.E.Y. Shapiro's Method, RR375, INRIA Rocq. Mars 1985. (to appear in the Journal of Logic Programming).

[Fra 85] G. Frandsen: A Denotational Semantics for Logic Programming. DAIMI PB 201, Aarhus University, Nov 1985.

[JM 84] N. D. Jones, A. Mycroft : Stepwise Development of Operational and Denotational Semantics for Prolog. Proc. 1984 Int. Symp. on Logic Programming, Atlantic City, N.J., 1984.

[Kee 85] R.A. O'Keefe : A Formal Definition of Prolog. Univ. of Auckland, BSI PS/22.

[Llo 84] J. W Lloyd : Foundations of Logic Programming. Springer Verlag, Berlin, 1984.

[Mos 81] C.D.S. Moss : The Formal Description of Programming Languages using Predicate Logic. Imperial College. DCS-July 1981, (BSI PS/37).

[MR 86] A. Martelli, G. Rossi: On the Semantics of Logic Programming Languages. Third Int. Conf. on Logic Programming, London, Jul. 1986. LNCS 225, pp 327-334.

[Nor 86] N.D. North: PROLOG A denotational Definition. National Laboratory, BSI-IST/5/15, PS141, Sept 1986.

[PWB 84] F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira: C-Prolog User's Manual. SRI International, Calif., Fev 1984.

[Rob 65] J. A. Robinson : A machine oriented logic based on the resolution principle. JACM 12, 1.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique